Threading on Mac OS X

Session 124

These are confidential sessions—please refrain from filming or taking pictures.

Application Technologies

Threading on Mac OS X

Mark Tozer
Hardware Evangelist

http://developer.apple.com/wwdc2005/

Threading on Mac OS X

George Warner
Schizophrenic Optimization Scientist

http://developer.apple.com/wwdc2005/

Agenda

- Threading terminology
- Why?
- Why not?
- Threading architectures
- Do’s and don’ts
- How to write Thread-Safe code
- Demos
- Q&A

Threading Terminology

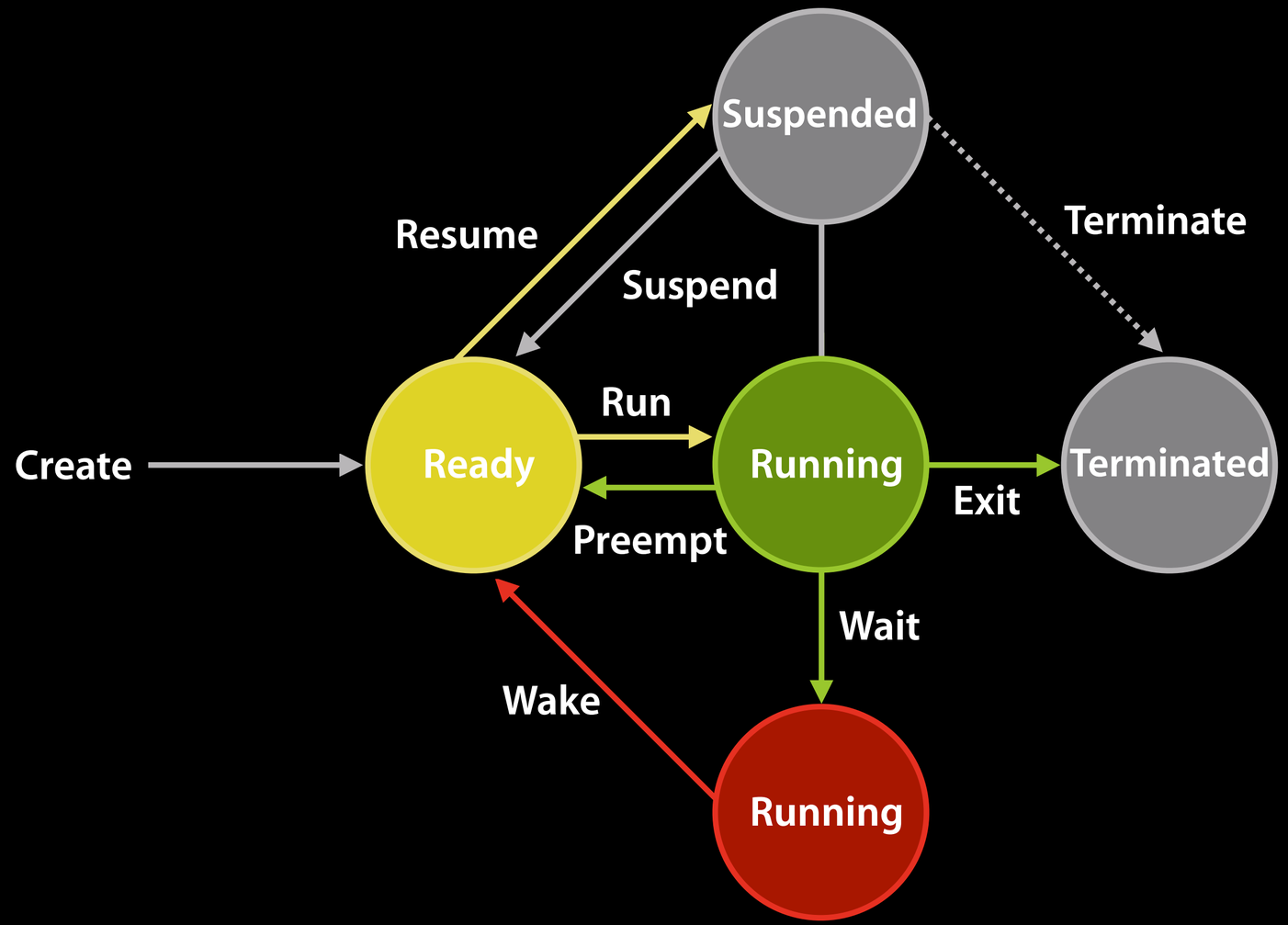
- Threads vs. processes
  - A thread is an independent execution path preemptively or cooperatively scheduled
  - A process is a collection of threads plus the resources necessary to run those threads
- Multiprocessing is a general case of multitasking
- Reentrant: concurrently executing the same code more than once
- Parallelism is a subset of concurrency

Threading Terminology

Thread State

- Ready
- Running
- Blocked (waiting)
- Terminated
- Suspended

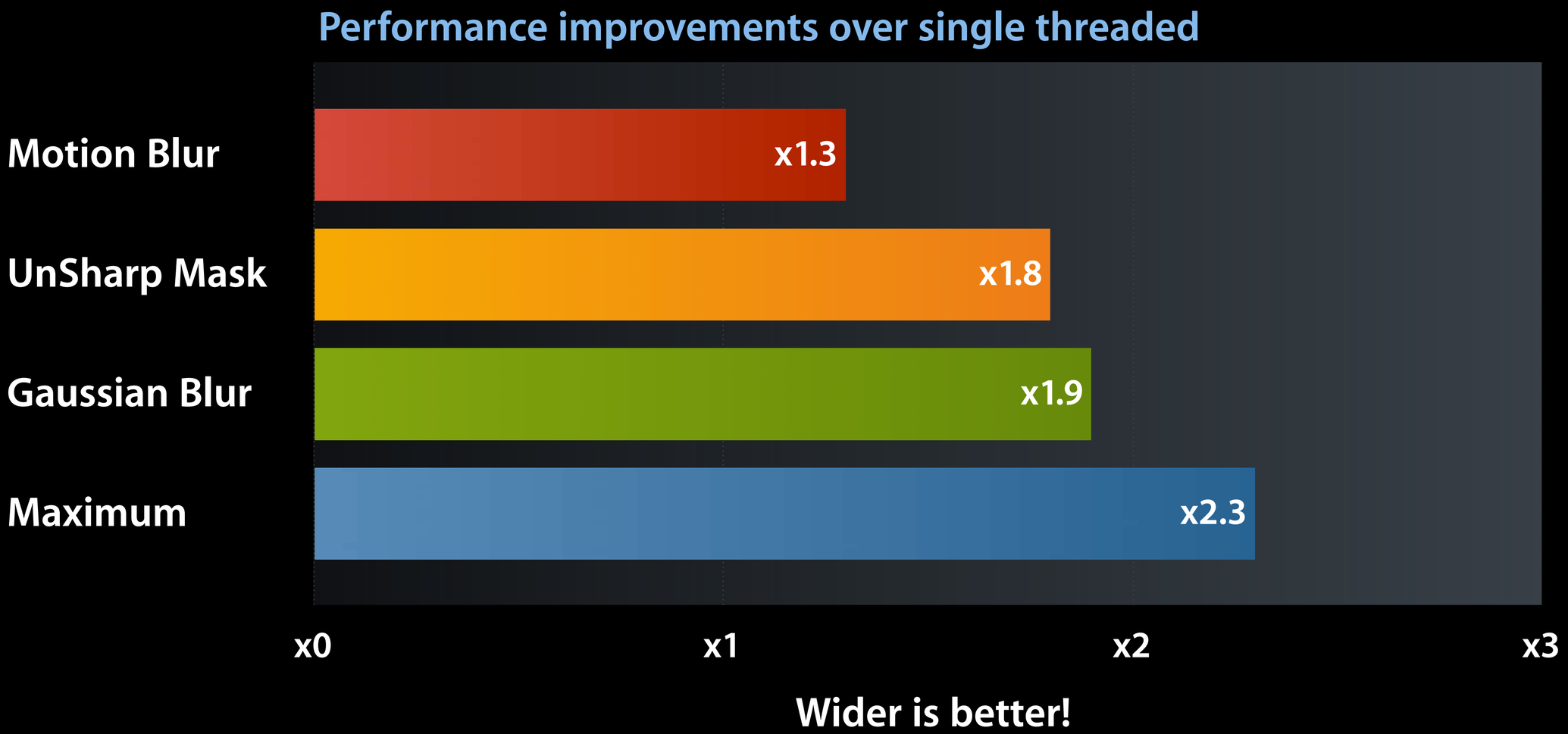
Threading Terminology



Why Use Threads?

- Customer expectations
  - Responsive GUI
  - No spinning beach ball!
- Scalability
  - MP hardware equals MP performance!

Scalability



Why Use Threads?

- Preemption
  - More threads equals more CPU time
- Synchronous requests
  - Put blocking calls into their own thread
- Polling is bad
  - Block Instead

Why Not?

- Added overhead
- Memory
- Added complexity
- 100s of threads
- Other options

Why Not?

- Added overhead
  - Creation time
    - 126 µSec (2.7 GHz DP G5)
  - Preemption time
    - 30 µSec (2.7 GHz DP G5)
  - Time quorum
    - 10 µSec (100 per second)

Why Not?

Memory

- Kernel resources (≈ 2K per thread)
  - Hardware context storage
  - 32 general-purpose registers
  - 32 floating-point registers
  - FP status and control register
  - XER, link and count registers
  - Memory management registers
  - Velocity Engine registers
- User-land resources (Carbon, Cocoa, etc.)
  - 512 KByte default (virtual) stack

Why Not?

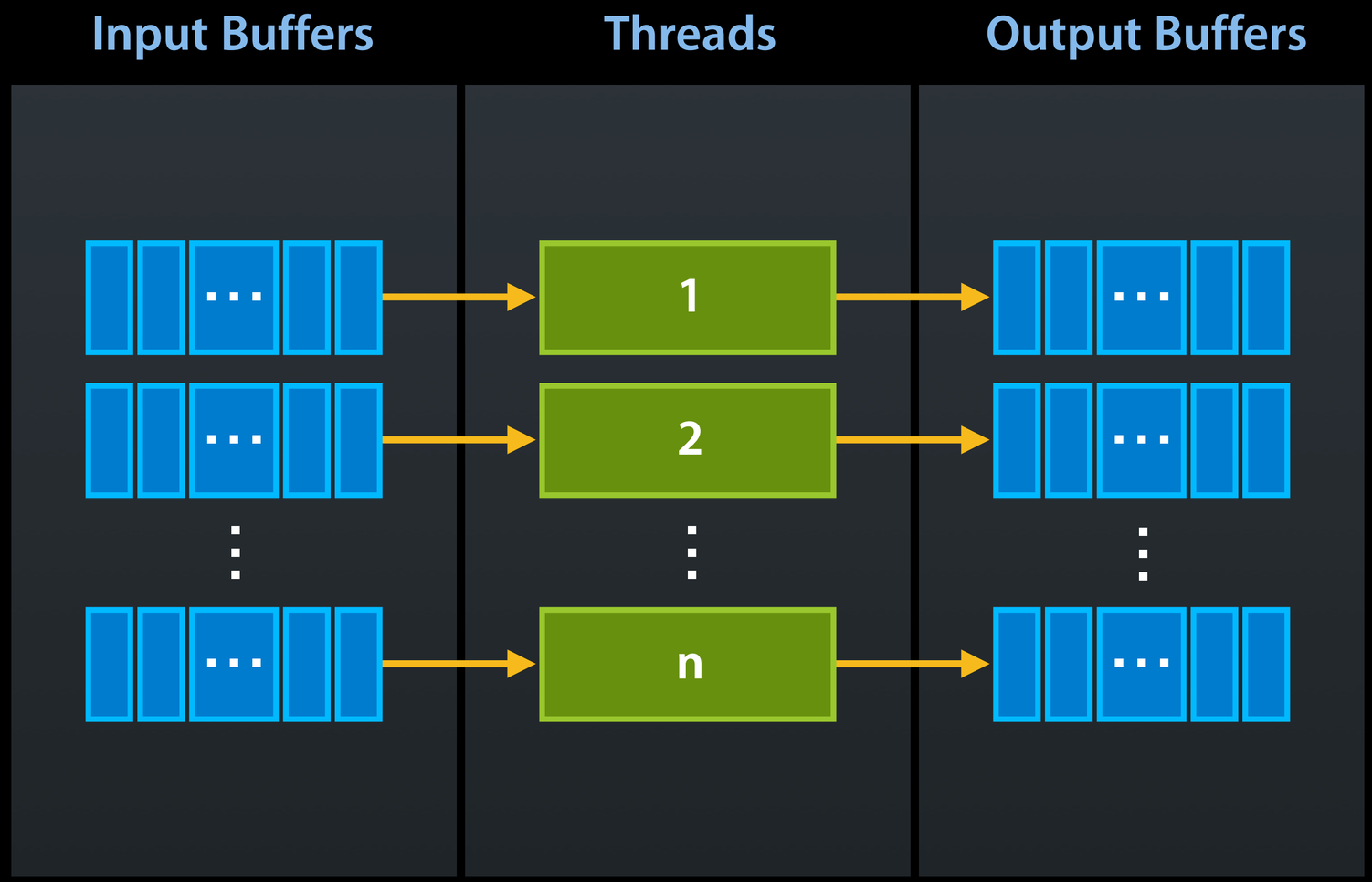
- Added complexity
  - Shared data may require locks
  - Non thread-safe API's
- 100s of threads
- Other options
  - Cooperative threads
  - Timers

Threading Architectures

- Parallel threads with parallel I/O buffers
- Parallel threads with shared I/O buffers
- Sequential threads with multiple I/O buffers

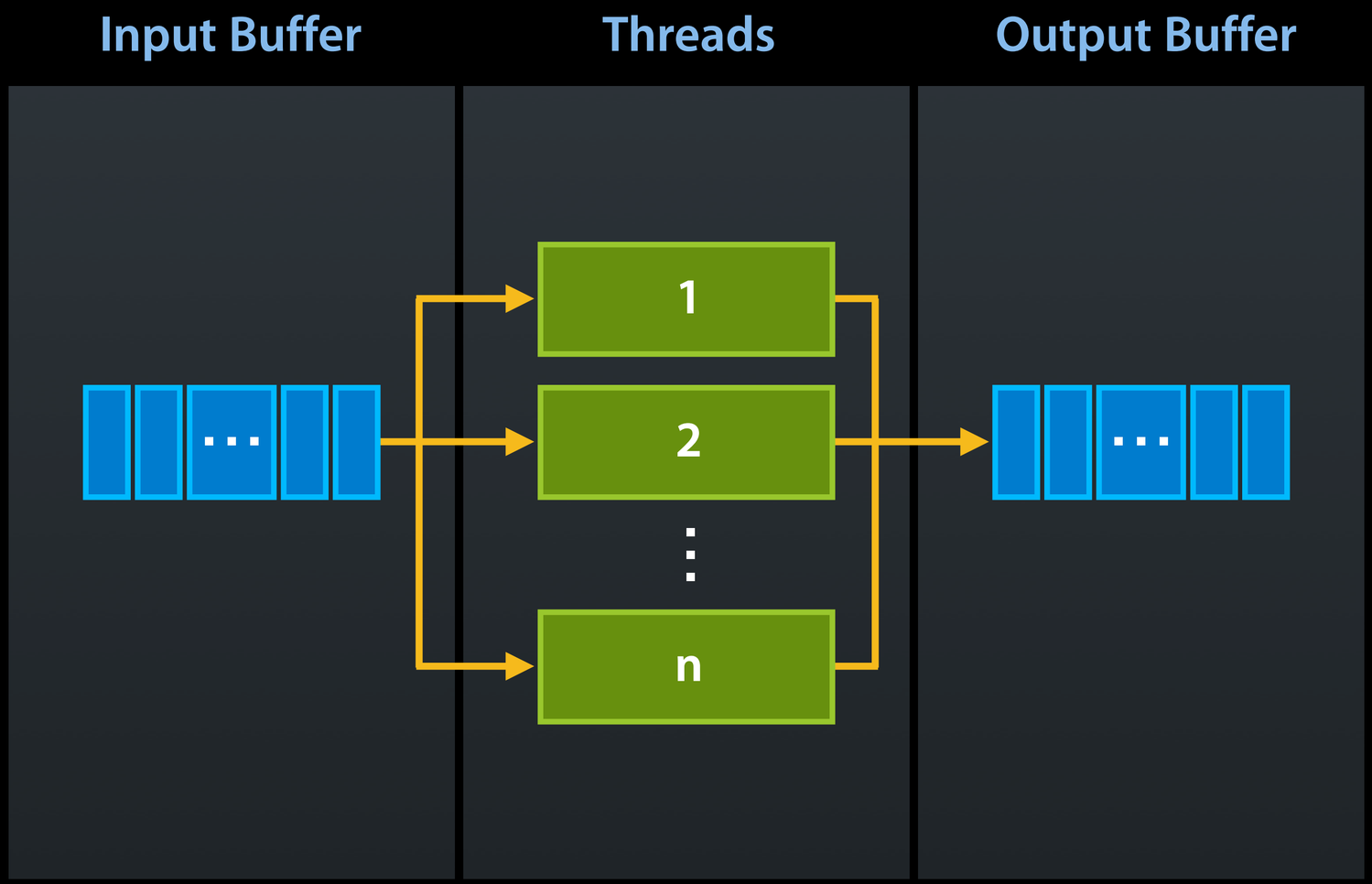
Threading Architectures

Parallel Threads with Parallel I/O Buffers



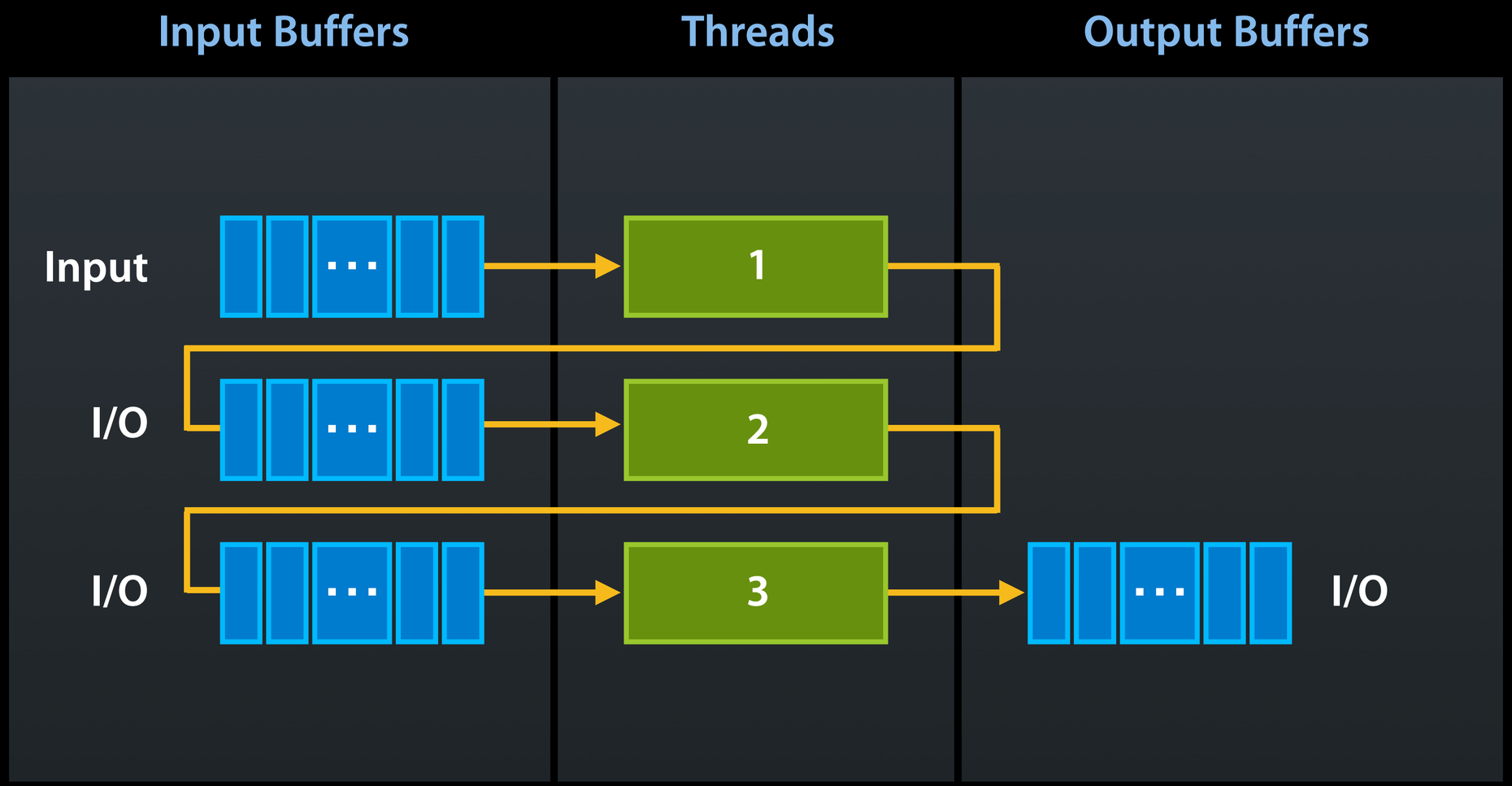
Threading Architectures

Parallel Threads with Shared I/O Buffers



Threading Architectures

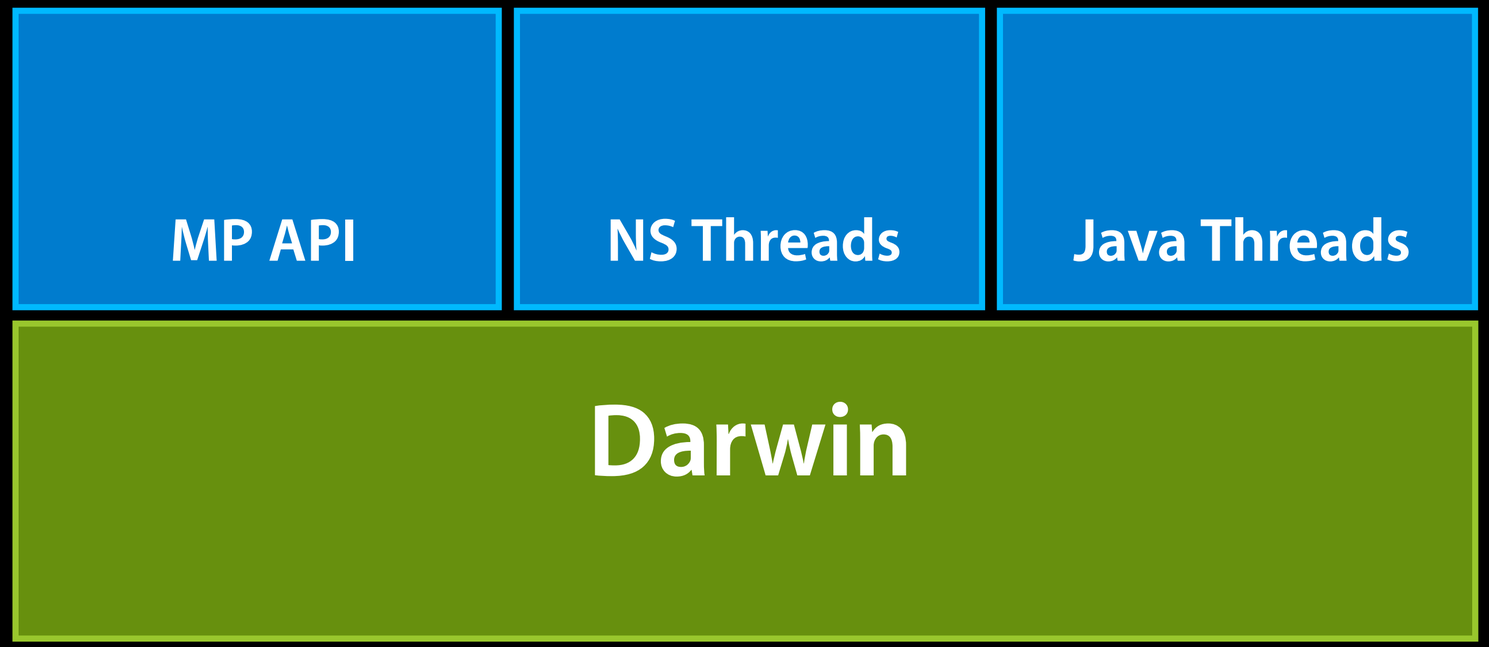
Sequential Threads with Multiple I/O Buffers



Threading Architectures

- Many applications have parallel and independent execution paths
- Examples:
  - A driving simulator (AI, physics, rendering, etc.)
  - An image processor (color correction, filters, fx's, etc.)
  - A word processor (grammar, spellcheck, kerning, etc.)

Threading Implementations



Threading Implementations

Common concepts

- Thread management
- Synchronization primitives
- Thread-safe services

Do’s and Don’ts

- Do avoid creation and destruction overhead
- Do be data-driven
- Don’t suspend, resume, or terminate
- Don’t under or over lock
- Don’t spin wait
- Don’t GUI

Do's and Don'ts

- Do avoid creation and destruction overhead
  - Pre-allocate and use pools
- Do be data-driven
  - Prologue (open, allocate, retain, etc.)
  - CRUNCH! (read, modify, write, etc.)
  - Epilogue (close, dispose, release, etc.)
- Don't suspend, resume, or terminate
  - Use synchronization methods instead

Do’s and Don’ts

- Don’t over or under lock
  - Find a balance
- Don’t spin wait
  - Use separate threads to merge signals
- Don’t GUI
  - Quartz OK
  - OpenGL OK
  - Use `PostEventToQueue`

Writing Thread-Safe Code

- Thread-safe code:
  - May be safely invoked concurrently by multiple threads
  - Can be called from multiple threads without unwanted interaction
  - Is reentrant or protected from multiple simultaneous execution by some form of mutual exclusion
  - Is guaranteed to compute the same result regardless of whether it it is run on one thread or many
- Or, thread-safe code:
  - Must behave correctly in a single-threaded environment
  - Must behave the same in a multi-threaded environment

Writing Thread-Safe Code

- Thread-safe
- Immutable
- Thread-aware
- Thread-compatible (friendly)
- Conditionally thread-safe
- Thread-hostile

Writing Thread-Safe Code

- Thread-safe
  - Can be executed in multiple threads without undesirable interactions
- Immutable
  - Guaranteed thread-safe
  - Never requires synchronization
- Thread-aware
  - Can detect when being executed in a thread
  - Uses locks to protect thread-unsafe code

Writing Thread-Safe Code

- Conditionally thread-safe
  - Specific usages are known non-thread-safe
  - Only thread-safe if executed once (to completion) before other threads call it
- Thread-compatible (friendly)
  - May require external synchronization
- Thread-hostile

Writing Thread-Safe Code

Things that are thread-safe:

- Immutable variables (any scope)
  - (Read-only data should be declared `const`)
- Non-shared data:
  - Local variables
  - Method parameters
  - Return values
  - Locally allocated memory

Writing Thread-Safe Code

Thread-safe code:

- May call other thread-safe functions
- May modify non-shared data
- May simultaneous read shared immutable data
- Must ensure exclusive access to any data it modifies
  - May simultaneous access distinct data

Writing Thread-Safe Code

Thread-safe code:

- Has no race conditions
- Does not deadlock
- Has no priority failures
- Has no starvation failures

Writing Thread-Safe Code

Static or global read/write data

- Eliminate
- Convert to thread-specific data
- Protect via synchronization methods

Writing Thread-Safe Code

Non-reentrant (non-thread safe) function

- `strcmp(char_to_upper(str1), char_to_upper(str2));`

```
char* char_to_upper(const char* string) {
  static char buffer[MAX_STRING_SIZE];

  for (int idx = 0; string[idx]; idx++) {
    buffer[idx] = toupper(string[idx]);
  }
  buffer[idx] = 0 // null terminate
  return buffer;
}
```

Writing Thread-Safe Code

Reentrant function (better)

- Requires error checking/external free

```
char* char_to_upper(const char* string) {
  char *buffer = malloc(MAX_STRING_SIZE);

  for (int idx = 0; string[idx]; idx ++) {
    buffer[idx] = toupper(string[idx]);
  }
  buffer[idx] = 0

  return buffer;
}
```

Writing Thread-Safe Code

Reentrant function (best!)

```
char* char_to_upper(const char* inStr, char*
outStr) {
  for (int idx = 0; inStr[idx]; idx ++) {
    outStr[idx] = toupper(inStr[idx]);
  }
  outStr[idx] = 0
  return outStr;
}
```

Writing Thread-Safe QuickTime Code

- QuickTime programming involves plug-in components
  - Can’t always predict which components you’ll use at run-time
  - Opening a movie may involve many components under the hood
  - Some components written by third parties, some legacy
  - How do you know whether the components are thread-safe?
  - Non-thread-safe components cause hard-to-reproduce crashes!
- `EnterMoviesOnThread`
  - Protects you from opening non-thread-safe components
  - High-level operations will return `componentNotThreadSafeErr`
  - May have to migrate work to the main thread

Writing Thread-Safe QuickTime Code

- When you call any NewMovie* API, the movie is attached to the current thread
  - If you want to use that movie from another thread, you must coordinate with QuickTime
  - `DetachMovieFromCurrentThread`
  - `AttachMovieToCurrentThread`
  - Important: check for errors returned by these APIs!
    - They may fail if the movie uses non-thread-safe components

- **See Technote 2125 for more info**

Demo

http://developer.apple.com/wwdc2005/

Documentation, Sample Code, and Other Resources

http://developer.apple.com/wwdc2005/

Who to Contact

George Warner
Schizophrenic Optimization Scientist
geowar@apple.com

Mark Tozer
Hardware Evangelist
tozer@apple.com

Multitasking / Symmetrical Multiprocessing Mailing List
<http://lists.apple.com/mailman/listinfo/mt-smp>

Q&A

George Warner
Schizophrenic Optimization Scientist

http://developer.apple.com/wwdc2005/

WWDC2005

Worldwide Developers Conference